服务器渲染的前世今生

对于渲染一词，大家都清楚，无非就是在浏览器上面渲染出页面。但是这里说的渲染和大家认知的渲染可不大一样。
这里先阐明一个公式：
页面 = 数据 + 模板
上面的加号（+）并不是加法，它就是我们今天要讲到的渲染。
服务端渲染：渲染过程在服务器端完成，最终的渲染结果 HTML 页面通过 HTTP 协议发送给客户端。对于客户端而言，只是看到了最终的 HTML 页面，看不到数据，也看不到模板。

几年前，几乎所有网站都使用 ASP、Java、PHP 这类做后端渲染，但后来随着 jQuery、Angular、React、Vue 等 JS 框架的崛起，开始转向了前端渲染。从 2014 年起又开始流行了同构渲染，号称是未来，集成了前后端渲染的优点，但转眼间三年过去了，很多当时壮心满满的框架（rendr、Lazo）从先驱变成了先烈。同构到底是不是未来？自己的项目该如何选型？我想不应该只停留在追求热门和拘泥于固定模式上，忽略了前后端渲染之“争”的“核心点”，关注如何提升“用户体验”。

> 明确三个概念：「后端渲染」指传统的 ASP、Java 或 PHP 的渲染机制；「前端渲染」指使用 JS 来渲染页面大部分内容，代表是现在流行的 SPA 单页面应用；「同构渲染」指前后端共用 JS，首次渲染时使用 Node.js 来直出 HTML。一般来说同构渲染是介于前后端中的共有部分。

前端渲染的优势

- 局部刷新。无需每次都进行完整页面请求
- 懒加载。如在页面初始时只加载可视区域内的数据，滚动后rp加载其它数据，可以通过 react-lazyload 实现
- 富交互。使用 JS 实现各种酷炫效果
- 节约服务器成本。省电省钱，JS 支持 CDN 部署，且部署极其简单，只需要服务器支持静态文件即可
- 天生的关注分离设计。服务器来访问数据库提供接口，JS 只关注数据获取和展现
- JS 一次学习，到处使用。可以用来开发 Web、Serve、Mobile、Desktop 类型的应用

后端渲染的优势

- 服务端渲染不需要先下载一堆 js 和 css 后才能看到页面（首屏性能）

- SEO
- 服务端渲染不用关心浏览器兼容性问题（随着浏览器发展，这个优点逐渐消失）
- 对于电量不给力的手机或平板，减少在客户端的电量消耗很重要

以上服务端优势其实只有首屏性能和 SEO 两点比较突出。但现在这两点也慢慢变得微不足道了。React 这类支持同构的框架已经能解决这个问题，尤其是 Next.js 让同构开发变得非常容易。还有静态站点的渲染，但这类应用本身复杂度低，很多前端框架已经能完全囊括。

大家对前端和后端渲染的现状基本达成共识。即前端渲染是未来趋势，但前端渲染遇到了首屏性能和SEO的问题。对于同构争议最多，在此我归纳一下。

前端渲染遇到的问题

前端渲染主要面临的问题有两个 **SEO**、**首屏性能**。
SEO 很好理解。由于传统的搜索引擎只会从 HTML 中抓取数据，导致前端渲染的页面无法被抓取。前端渲染常使用的 SPA 会把所有 JS 整体打包，无法忽视的问题就是文件太大，导致渲染前等待很长时间。特别是网速差的时候，让用户等待白屏结束并非一个很好的体验。

同构的优点

同构恰恰就是为了解决前端渲染遇到的问题才产生的，至 2014 年底伴随着 React 的崛起而被认为是前端框架应具备的一大杀器，以至于当时很多人为了用此特性而放弃 Angular 1 而转向 React。然而近3年过去了，很多产品逐渐从全栈同构的理想化逐渐转到首屏或部分同构。让我们再一次思考同构的优点真是优点吗?

1. 有助于 SEO

首先确定你的应用是否都要做 SEO，如果是一个后台应用，那么只要首页做一些静态内容宣导就可以了。如果是内容型的网站，那么可以考虑专门做一些页面给搜索引擎
时到今日，谷歌已经能够可以在爬虫中执行 JS 像浏览器一样理解网页内容，只需要往常一样使用 JS 和 CSS 即可。并且尽量使用新规范，使用 pushstate 来替代以前的 hashstate。不同的搜索引擎的爬虫还不一样，要做一些配置的工作，而且可能要经常关注数据，有波动那么可能就需要更新。第二是该做 sitemap 的还得做。相信未来即使是纯前端渲染的页面，爬虫也能很好的解析。

2. 共用前端代码，节省开发时间

其实同构并没有节省前端的开发量，只是把一部分前端代码拿到服务端执行。而且为了同构还要处处兼容 Node.js 不同的执行环境。有额外成本，这也是后面会具体谈到的。

3. 提高首屏性能

由于 SPA 打包生成的 JS 往往都比较大，会导致页面加载后花费很长的时间来解析，也就造成了白屏问题。服务端渲染可以预先使到数据并渲染成最终 HTML 直接展示，理想情况下能避免白屏问题。在我参考过的一些产品中，很多页面需要获取十几个接口的数据，单是数据获取的时候都会花费数秒钟，这样全部使用同构反而会变慢。

同构并没有想像中那么美

1. 性能

把原来放在几百万浏览器端的工作拿过来给你几台服务器做，这还是花挺多计算力的。尤其是涉及到图表类需要大量计算的场景。这方面调优，可以参考 walmart的调优策略。
个性化的缓存是遇到的另外一个问题。可以把每个用户个性化信息缓存到浏览器，这是一个天生的分布式缓存系统。我们有个数据类应用通过在浏览器合理设置缓存，双十一当天节省了 70% 的请求量。试想如果这些缓存全部放到服务器存储，需要的存储空间和计算都是很非常大。

2. 不容忽视的服务器端和浏览器环境差异

前端代码在编写时并没有过多的考虑后端渲染的情景，因此各种 BOM 对象和 DOM API 都是拿来即用。这从客观层面也增加了同构渲染的难度。我们主要遇到了以下几个问题：

- document 等对象找不到的问题
- DOM 计算报错的问题
- 前端渲染和服务端渲染内容不一致的问题

由于前端代码使用的 `window` 在 node 环境是不存在的，所以要 mock window，其中最重要的是 cookie，userAgent，location。但是由于每个用户访问时是不一样的 `window`，那么就意味着你得每次都更新 `window`。
而服务端由于 js require 的 cache 机制，造成前端代码除了具体渲染部分都只会加载一遍。这时候 `window` 就得不到更新了。所以要引入一个合适的更新机制，比如把读取改成每次用的时候再读取。

```
export const isSsr = () => (
  !(typeof window !== 'undefined' && window.document && window.document.createElement
&& window.setTimeout)
);
```

原因是很多 DOM 计算在 SSR 的时候是无法进行的，涉及到 DOM 计算的的内容不可能做到 SSR 和 CSR 完全一致，这种不一致可能会带来页面的闪动。

3. 内存溢出

前端代码由于浏览器环境刷新一遍内存重置的天然优势，对内存溢出的风险并没有考虑充分。
比如在 React 的 `componentWillMount` 里做绑定事件就会发生内存溢出，因为 React 的设计是后端渲染只会运行 `componentDidMount` 之前的操作，而不会运行 `componentWillUnmount` 方法（一般解绑事件在这里）。

4. 异步操作

前端可以做非常复杂的请求合并和延迟处理，但为了同构，所有这些请求都在预先拿到结果才会渲染。而往往这些请求是有很多依赖条件的，很难调和。纯 React 的方式会把这些数据以埋点的方式打到页面上，前端不再发请求，但仍然再渲染一遍来比对数据。造成的结果是流程复杂，大规模使用成本高。幸运的是 Next.js 解决了这一些，后面会谈到。

5. simple store（redux）

这个 store 是必须以字符串形式塞到前端，所以复杂类型是无法转义成字符串的，比如function。
总的来说，同构渲染实施难度大，不够优雅，无论在前端还是服务端，都需要额外改造。

首屏优化

再回到前端渲染遇到首屏渲染问题，除了同构就没有其它解法了吗？总结以下可以通过以下三步解决

1. 分拆打包

现在流行的路由库如 react-router 对分拆打包都有很好的支持。可以按照页面对包进行分拆，并在页面切换时加上一些 loading 和 transition 效果。

2. 交互优化

打开渲染并没有白屏，有两段加载动画，第一段像是加载资源，第二段是一个加载占位器，过去我们会用 loading 效果，但过渡性不好。近年流行 Skeleton Screen（加载占位符） 效果。其实就是在白屏无法避免的时候，为了解决等待加载过程中白屏或者界面闪烁造成的割裂感带来的解决方案。

3. 部分同构

部分同构可以降低成功同时利用同构的优点，如把核心的部分如菜单通过同构的方式优先渲染出来。我们现在的做法就是使用同构把菜单和页面骨架渲染出来。给用户提示信息，减少无端的等待时间。

相信有了以上三步之后，首屏问题已经能有很大改观。相对来说体验提升和同构不分伯仲，而且相对来说对原来架构破坏性小，入侵性小。是我比较推崇的方案。

我们赞成客户端渲染是未来的主要方向，服务端则会专注于在数据和业务处理上的优势。但由于日趋复杂的软硬件环境和用户体验更高的追求，也不能只拘泥于完全的客户端渲染。同构渲染看似美好，但以目前的发展程度来看，在大型项目中还不具有足够的应用价值，但不妨碍部分使用来优化首屏性能。做同构之前 ，一定要考虑到浏览器和服务器的环境差异，站在更高层面考虑。